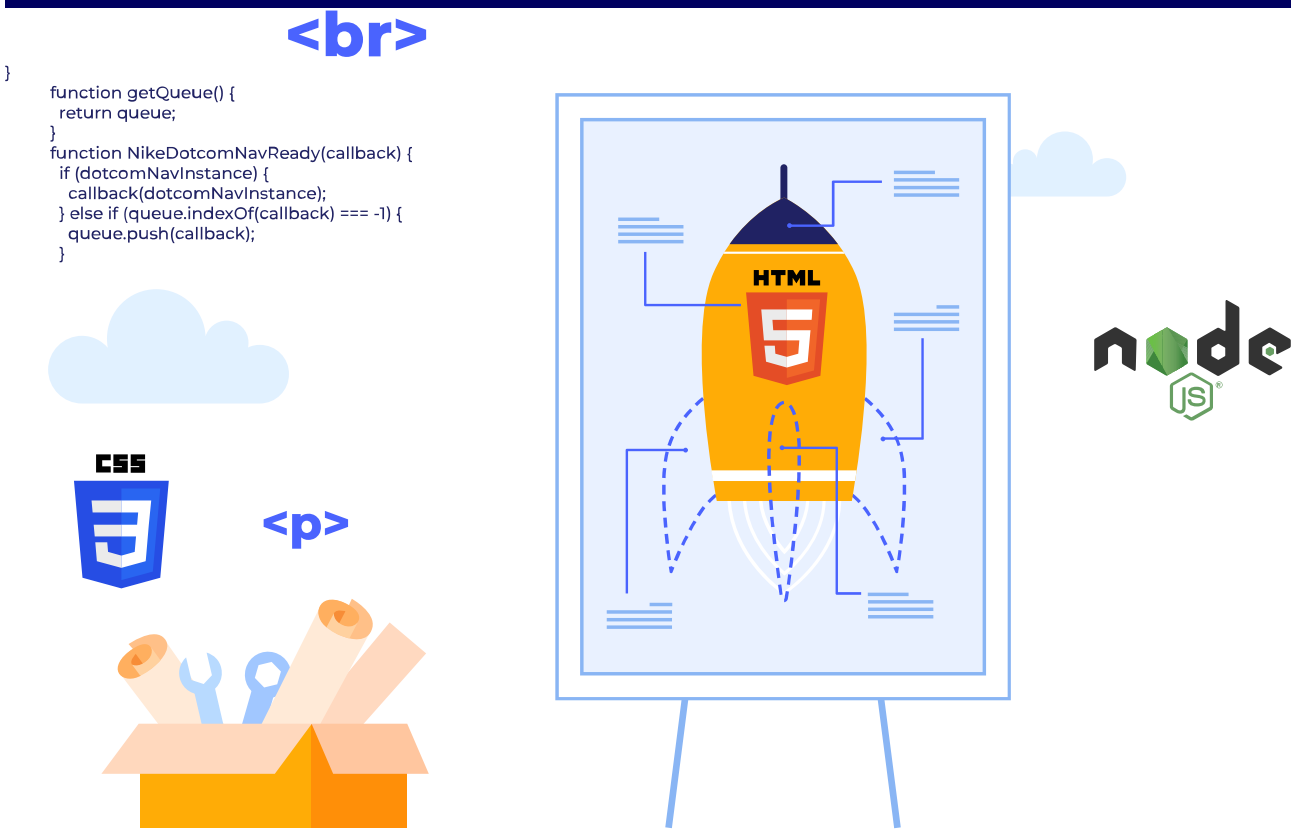
Programación Web



José Luis Quiroz Fabián

Índice

1 TypeScript

TypeScript es un lenguaje que permite tener una orientación a objetos clara en JavaScript, además añade un tipado fuerte y es el lenguaje que utilizamos para programar aplicaciones web con Angular 2 que es uno de los frameworks más populares para desarrollar aplicaciones modernas y escalables en el lado del cliente. TypeScript se puede considerar un superconjunto de JavaScript. TypeScript es un lenguaje libre desarrollado por Microsoft y es un superconjunto de JavaScript que gracias a las ventajas que ofrece está siendo cada vez más utilizado.

1.1 Instalación

Para la instalación de TypeScript se requiere NodeJS. Teniendo instalado NodeJs basta con ejecutar:

```
usuario$ npm install -g typescript
```

Para verificar y obtener la versión de typescript se puede ejecutar:

```
usuario$ tsc -v
Version 3.9.5
```

En este ejemplo nos indica que la versión instalada es la 3.9.5.

1.2 Primer código: Saludos

Los navegadores no reconocen el lenguaje TypeScript, por lo que se requiere hacer una traducción de TypeScript a JavaScript. Para lo anterior, considere del siguiente ejemplo:

```
1  <html lang="es">
2
3  <head>
4      <meta charset="UTF-8">
5      <title>Introducción</title>
6  </head>
7
8  <body>
9
10     <script src="app.js"></script>
11 </body>
12
13 </html>
```

Genere el archivo de TypeScript llamado *app.ts* con el siguiente código:

```
(()=>{

  function getMensaje(nombre){

    return "Hola: "+nombre;
  }

  console.log(getMensaje("Luis"));


})();
```

Para generar el código JavaScript (el archivo *app.js*) se debe ejecutar:

```
usuario@pc:~$tsc app.ts
```

Lo anterior genera el archivo *app.js* con el siguiente contenido:

```
(function () {
    function getMensaje(nombre) {
        return "Hola: " + nombre;
    }
    console.log(getMensaje("Luis"));
})();
```

1.3 Otro ejemplo

Considere el siguiente ejemplo en TypeScript (app.ts):

```
(function(){
  function saludar( nombre:string ) {
      console.table( 'Hola, ' + nombre ); // Hola Logan
  }


  const profesor = {
      nombre: 'Luis'
  };


  saludar( profesor.nombre );
})();
```

Al generar el codigo JavaScript se obtiene lo siguiente:

```
(function () {
    function saludar(nombre) {
        console.table('Hola, ' + nombre); // Hola Logan
    }
    var profesor = {
        nombre: 'Luis'
    };
    saludar(profesor.nombre);
})();
```

PARA GENERAR EL ARCHIVO DE CONFIGURACIÓN DE TYPESCRIPT, EL CUAL INDICA COMO SE GENERARÁ EN ARCHIVO JAVASCRIPT SE EJECUTA:

```
usuario@pc:~$tsc --init
```

Como resultado se obtiene algo similar a:

```
{
  "compilerOptions": {
    /* Visit https://aka.ms/tsconfig.json to read more about this file */

    /* Basic Options */
    // "incremental": true,                   /* Enable incremental
       ↪ compilation */
    "target": "es5",                          /* Specify ECMAScript
       ↪ target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016',
       ↪ 'ES2017', 'ES2018', 'ES2019', 'ES2020', or 'ESNEXT'. */
    "module": "commonjs",                     /* Specify module code
       ↪ generation: 'none', 'commonjs', 'amd', 'system', 'umd',
       ↪ 'es2015', 'es2020', or 'ESNext'. */
    // "lib": [],                             /* Specify library files
       ↪ to be included in the compilation. */
    // "allowJs": true,                       /* Allow javascript files
       ↪ to be compiled. */
    // "checkJs": true,                       /* Report errors in .js
       ↪ files. */
    // "jsx": "preserve",                     /* Specify JSX code
       ↪ generation: 'preserve', 'react-native', or 'react'. */
    // "declaration": true,                   /* Generates corresponding
       ↪ '.d.ts' file. */
    // "declarationMap": true,                /* Generates a sourcemap
       ↪ for each corresponding '.d.ts' file. */
```

```
15      // "sourceMap": true,                    /* Generates corresponding
           ↪ '.map' file. */
16      // "outFile": "./",                      /* Concatenate and emit
           ↪ output to single file. */
17      // "outDir": "./",                       /* Redirect output
           ↪ structure to the directory. */
18      // "rootDir": "./",                      /* Specify the root
           ↪ directory of input files. Use to control the output directory
           ↪ structure with --outDir. */
19      // "composite": true,                    /* Enable project
           ↪ compilation */
20      // "tsBuildInfoFile": "./",              /* Specify file to store
           ↪ incremental compilation information */
21      // "removeComments": true,               /* Do not emit comments to
           ↪ output. */
22      // "noEmit": true,                       /* Do not emit outputs. */
23      // "importHelpers": true,                /* Import emit helpers
           ↪ from 'tslib'. */
24      // "downlevelIteration": true,           /* Provide full support
           ↪ for iterables in 'for-of', spread, and destructuring when
           ↪ targeting 'ES5' or 'ES3'. */
25      // "isolatedModules": true,              /* Transpile each file as
           ↪ a separate module (similar to 'ts.transpileModule'). */
26
27      /* Strict Type-Checking Options */
28      "strict": true,                          /* Enable all strict
           ↪ type-checking options. */
29      // "noImplicitAny": true,                /* Raise error on
           ↪ expressions and declarations with an implied 'any' type. */
30      // "strictNullChecks": true,             /* Enable strict null
           ↪ checks. */
31      // "strictFunctionTypes": true,          /* Enable strict checking
           ↪ of function types. */
32      // "strictBindCallApply": true,          /* Enable strict 'bind',
           ↪ 'call', and 'apply' methods on functions. */
33      // "strictPropertyInitialization": true,  /* Enable strict checking
           ↪ of property initialization in classes. */
34      // "noImplicitThis": true,               /* Raise error on 'this'
           ↪ expressions with an implied 'any' type. */
35      // "alwaysStrict": true,                 /* Parse in strict mode
           ↪ and emit "use strict" for each source file. */
```

```
36
37      /* Additional Checks */
38      // "noUnusedLocals": true,                  /* Report errors on unused
           ↪ locals. */
39      // "noUnusedParameters": true,              /* Report errors on unused
           ↪ parameters. */
40      // "noImplicitReturns": true,               /* Report error when not
           ↪ all code paths in function return a value. */
41      // "noFallthroughCasesInSwitch": true,      /* Report errors for
           ↪ fallthrough cases in switch statement. */
42
43      /* Module Resolution Options */
44      // "moduleResolution": "node",              /* Specify module
           ↪ resolution strategy: 'node' (Node.js) or 'classic' (TypeScript
           ↪ pre-1.6). */
45      // "baseUrl": "./",                         /* Base directory to
           ↪ resolve non-absolute module names. */
46      // "paths": {},                             /* A series of entries
           ↪ which re-map imports to lookup locations relative to the
           ↪ 'baseUrl'. */
47      // "rootDirs": [],                          /* List of root folders
           ↪ whose combined content represents the structure of the project
           ↪ at runtime. */
48      // "typeRoots": [],                         /* List of folders to
           ↪ include type definitions from. */
49      // "types": [],                             /* Type declaration files
           ↪ to be included in compilation. */
50      // "allowSyntheticDefaultImports": true,  /* Allow default imports
           ↪ from modules with no default export. This does not affect code
           ↪ emit, just typechecking. */
51      "esModuleInterop": true,                    /* Enables emit
           ↪ interoperability between CommonJS and ES Modules via creation
           ↪ of namespace objects for all imports. Implies
           ↪ 'allowSyntheticDefaultImports'. */
52      // "preserveSymlinks": true,                /* Do not resolve the real
           ↪ path of symlinks. */
53      // "allowUmdGlobalAccess": true,            /* Allow accessing UMD
           ↪ globals from modules. */
54
55      /* Source Map Options */
56      // "sourceRoot": "",                        /* Specify the location
```

```
        ↪ where debugger should locate TypeScript files instead of
        ↪ source locations. */
57      // "mapRoot": "",                       /* Specify the location
        ↪ where debugger should locate map files instead of generated
        ↪ locations. */
58      // "inlineSourceMap": true,             /* Emit a single file with
        ↪ source maps instead of having a separate file. */
59      // "inlineSources": true,               /* Emit the source
        ↪ alongside the sourcemaps within a single file; requires
        ↪ '--inlineSourceMap' or '--sourceMap' to be set. */
60
61      /* Experimental Options */
62      // "experimentalDecorators": true,      /* Enables experimental
        ↪ support for ES7 decorators. */
63      // "emitDecoratorMetadata": true,       /* Enables experimental
        ↪ support for emitting type metadata for decorators. */
64
65      /* Advanced Options */
66      "skipLibCheck": true,                   /* Skip type checking of
        ↪ declaration files. */
67      "forceConsistentCasingInFileNames": true  /* Disallow
        ↪ inconsistently-cased references to the same file. */
68    }
69  }
```

Para poner a TypeScript en modo de observador, basta con ejecutar:

```
tsc --watch
```

1.4 Variables y Constantes

En TypeScript las variables tienen un alcance y un tipo. En general la sintaxis para declarar una variable en TypeScript es:

```
alcance nombreVarable:tipoDeDato=valor;
```

1.4.1 Alcance

El alcance de la variable se define mediante los modificadores *let* y *var*. Las variables *var* tienen un alcance global y las variables *let* un alcance local. Para las constantes se utiliza *const*. Considere el siguiente ejemplo en TypeScript:

```

(function(){

    var mensaje = "Hola";

    const URL = "constantes!!!"

    let variable="Mexico";

    if(true){

        var mensaje = "Mundo";
        let variable="Lindo";

    }

    console.log(mensaje+" "+variable);

})();
```

El código JavaScript

```
"use strict";
(function () {
    var mensaje = "Hola";
    var URL = "constantes!!!";
    var variable = "Mexico";
    if (true) {
        var mensaje = "Mundo";
        var variable_1 = "Lindo";
    }
    console.log(mensaje + " " + variable);
})();
```

1.4.2 Tipos de datos

En TypeScript existen diferentes tipos de datos para las variables y estos se deben especificar cuando se declara la variable o cuando se le asigna de inicio un valor, el tipo de la variable se asigna en base al valor de inicialización. No esta de más mencionar que los tipos de las variables se deben respetar. Cuando a una variable no se le indica su tipo es de tipo *any*. Una variable *any* puede tomar el valor de cualquier variable o bien de un subgrupo de tipos. Ejemplo:

```

```

```
(function(){

    //Tipo string
    let mensaje:string='Hola'
    //Tipo numero
    let numero:number =123;
    //Tipo boolean
    let booleano:boolean=true;
    let dia:Date=new Date();
    //Tipo any, un tipo especial
    let otra;
    //Se le puede asignar cualquier cosa a any
    otra=mensaje;
    otra=numero;
    //Se le puede asigar solo cadenas y enteros
    let otro:string|number;
    otro = mensaje;
    otro = numero
    let punto={

        x:0,
        y:0
    }

    punto={
        x:6,
        y:6

    }

    console.log(punto);

})();
```

El código JavaScript

```
"use strict";
(function () {
    //Tipo string
    var mensaje = 'Hola';
    //Tipo numero
    var numero = 123;
    //Tipo boolean
```

```
8     var booleano = true;
9     var dia = new Date();
10    //Tipo any, un tipo especial
11    var otra;
12    //Se le puede asignar cualquier cosa a any
13    otra = mensaje;
14    otra = numero;
15    //Se le puede asigar solo cadenas y enteros
16    var otro;
17    otro = mensaje;
18    otro = numero;
19    var punto = {
20        x: 0,
21        y: 0
22    };
23    punto = {
24        x: 6,
25        y: 6
26    };
27    console.log(punto);
28 })();
```

1.5 Funciones

En las funciones de TypeScript se debe especificar el tipo de dato que se recibe como parámetro y de forma opcional el tipo de dato que regresará la función.

```
1
2  (()=>{
3
4      function factorial(n:number):number{
5
6          let i:number;
7          let fact:number=1;
8          for(i=1;i<=n;i++)
9              fact=fact*i;
10         return fact;
11     }
12
13     console.log(factorial(5));
14
15 })();
```

Además, es posible indicar que unos parámetros son opcionales. Por ejemplo, en el código de abajo el parámetro *x* de la función *exponencial* (línea 24) es opcional. Cuando este parámetro no se recibe dentro de la función se la asigna el valor *1* (tal es el caso de la invocación en la línea 39). En la invocación de la línea 40 este parámetro si aparece, por lo que el valor de *x* es *2*. Además, es posible tener un parámetro opcional sin un valor por defecto, tal es el caso del parámetro *n*, el cual se puede o no recibir (ver la invocación en la línea 39 y 40).

```
1
2  (()=>{
3
4
5      function factorial(n:number):number{
6
7          let i:number;
8          let fact:number=1;
9          for(i=1;i<=n;i++)
10             fact=fact*i;
11         return fact;
12     }
13
14     function potencia(a:number,b:number):number{
15
16         let i:number;
17         let pot:number=1;
18         for(i=1;i<=b;i++)
19             pot=pot*a;
20         return pot;
21     }
22
23
24     function exponencial(x:number=1,n?:number):number{
25
26         let i:number=1;
27         let iteraciones:number,suma:number=0;
28         if(n){
29             iteraciones=n;
30         }else
31             iteraciones=5;
32         for(i=0;i<=iteraciones;i++){
33             suma=suma+potencia(x,i)/factorial(i);
34         }
35         return suma;
```

```
36
37      }
38
39      console.log(exponencial())
40      console.log(exponencial(2))
41      console.log(exponencial(2,10))
42
43  })();
```

1.5.1 Funciones de flecha

Una expresión de función flecha tiene una sintaxis más compacta que una expresión de función regular, por lo que son una buena alternativa a estas últimas. Una diferencia entre las funciones flecha y las funciones tradicionales en JavaScript es que no tienen su propio *this*.

```
1
2   (function(){
3
4       const f = function(s:string){
5
6           return s.toUpperCase();
7
8       }
9
10      const g = (s:string)=> s.toUpperCase();
11
12
13      const sumar_js = function(a:number,b:number){
14
15          return a+b;
16
17      }
18
19      const sumar_ts = (a:number,b:number)=>a+b;
20
21
22      const persona ={
23
24          nombre:'JL',
25          getNombre(){
26
27              setTimeout(()=>{
```

```
28
29                  console.log(`${this.nombre} ok!!!!`);
30
31              },1000);
32
33
34          }
35
36      }
37      persona.getNombre();
38      console.log(g("Hola"));
39  })();
```

El código JavaScript del código de arriba:

```
1   (function () {
2       var f = function (s) {
3           return s.toUpperCase();
4       };
5       var g = function (s) { return s.toUpperCase(); };
6       var sumar_js = function (a, b) {
7           return a + b;
8       };
9       var sumar_ts = function (a, b) { return a + b; };
10      var persona = {
11          nombre: 'JL',
12          getNombre: function () {
13              var _this = this;
14              setTimeout(function () {
15                  console.log(_this.nombre + " ok!!!!");
16              }, 1000);
17          }
18      };
19      persona.getNombre();
20      console.log(g("Hola"));
21  })();
```

1.6 Template literales

Permiten manipular las cadenas para formatear su contenido. Por ejemplo:

```
1
2   (function(){
```

```
    const nombre = "Luis";
    const apellido ="Aguilar";
    const edad =33;

    function getEdad(){

        return 100+200;
    }

    let salida = `${nombre} ${apellido}(${edad})`;

    console.log(salida)

    salida = `
    ${nombre}
    ${apellido}
    (${edad})`;

    console.log(salida)

    salida = `El nombre es ${nombre}$ y su apellido ${apellido}(${edad
   +100})`;

    console.log(salida)

    salida = `${nombre}${apellido}(${getEdad()})`;

    console.log(salida)

})();
```

El código JavaScript

```
(function () {
    var nombre = "Luis";
    var apellido = "Aguilar";
    var edad = 33;
    function getEdad() {
        return 100 + 200;
    }
    var salida = nombre + " " + apellido + "(" + edad + ")";
    console.log(salida);
    salida = "\n  " + nombre + " \n  " + apellido + "\n  (" + edad + ")";
```

```
    console.log(salida);
    salida = "El nombre es " + nombre + "$ y su apellido " + apellido + "
   (" + (edad + 100) + ")";
    console.log(salida);
    salida = "" + nombre + apellido + "(" + getEdad() + ")";
    console.log(salida);
})();
```

Otro ejemplo:

```

(function(){



    function activar(quien:string, objeto: string="opcional1",
       otroopcional?:string){

        if(otroopcional)
           console.log(`${quien} recibio ${objeto} y llego ${otroopcional
              }`);
        else
           console.log(`${quien} recibio ${objeto}`);



    }

    activar("Luis","la máquina");
    activar("Luis","la máquina","tarde");
})();
```

El código JavaScript

```
(function () {
    function activar(quien, objeto, otroopcional) {
        if (objeto === void 0) { objeto = "opcional1"; }
        if (otroopcional)
            console.log(quien + " recibio " + objeto + " y llego " +
   otroopcional);
        else
            console.log(quien + " recibio " + objeto);
    }
    activar("Luis", "la máquina");
    activar("Luis", "la máquina", "tarde");
})();
```

1.7 Desestructuración

En TypeScript es posible descomponer un objeto en sus elementos y poder trabajar de forma aislada con esto elementos. Lo anterior ese conoce como la desestructuración. Por ejemplo, en el siguiente ejemplo se crea un objeto llamado gato con 3 atributos (lineas 5-11). Tradicionalmente como se hace en JavaScript para acceder a sus elementos se utiliza el nombre del objeto, seguido de un punto y después el nombre del atributo al que se desea acceder (lineas 12-14). La desestructuración del objeto gato se realiza en la línea 17: `const nombre,color,raza=gato;`. Lo anterior permite trabajar con cada atributo de forma aislada (líneas 19-21). De igual forma, la desestructuración se pude hacer en los parámetros de una función (líneas 24-30).

```
1
2  (()=>{
3
4
5   const gato={
6
7      nombre:"bigotes",
8      color:"cafe",
9      raza:"siamés"
10
11  }
12    console.log(gato.nombre);
13    console.log(gato.color);
14    console.log(gato.raza);
15
16
17    const {nombre,color,raza}=gato;
18
19    console.log(nombre);
20    console.log(color);
21    console.log(raza);
22
23
24  const extraer =({nombre,color}:any)=>{
25
26     console.log(nombre);
27     console.log(color);
28
29
30  }
31
```

```
32  extraer(gato);
33
34
35 })();
```

Lo anterior también funciona con arreglos, solo que en lugar de usar { se usan [. Lo anterior se puede ver en el siguiente ejemplo:

```
1
2  (()=>{
3
4
5   const modelos:string[]=["Rio","Sentra","Versa"];
6
7
8   const [,sentra,]=modelos;
9
10  console.log(sentra);
11
12
13 const getModelos =([kia,nissan,]:string[])=>{
14
15      console.log(kia);
16      console.log(nissan);
17
18 }
19
20 getModelos(modelos);
21
22 })();
```

En este ejemplo se define un arreglo llamado modelos (línea 5) y ese se desestructura en la línea 8 para acceder de forma simple en su entrada 2 (sentra). Además, se define una función la cual desestructura un arreglo que recibe para utilizar directamente los elementos 1 y 2 del arreglo respectivamente.

Este ultimo ejemplo en JavaScript se ve como se muestra en el siguiente código:

```
1 (function () {
2     var modelos = ["Rio", "Sentra", "Versa"];
3     var sentra = modelos[1];
4     console.log(sentra);
5     var getModelos = function (_a) {
6         var kia = _a[0], nissan = _a[1];
7         console.log(kia);
```

```
        console.log(nissan);
    };
    getModelos(modelos);
})();
```

1.8 Clases

Una clase es como una estructura predefinida que sirve como molde para crear Objetos. En este molde se declaran atributos y métodos; un atributo es una variable, y un método es una función. La forma de crear una clase en Typescript es usando la palabra reservada çlass"de la siguiente forma:

```
(()=>{

class MiClase {
}

})();
```

La forma de crear un nuevo objeto usando una clase, es utilizando la palabra reservada "new", como se muestra en el siguiente código:

```
(()=>{

let miVariable = new MiClase();

})();
```

1.8.1 Atributos y métodos

Los atributos son variables en la clase que definen el estado de los objetos. Los métodos son los comportamientos de la clase o de los objetos. Por ejemplo, al crear una clase Automovil con un atributo y dos métodos se pueden definir la estructura siguiente:

```
(()=>{

    class Automovil {

      color:string;

      getColor(){

         return this.color;
```

```
10
11        }
12        setColor(color:string){
13
14            this.color=color;
15
16        }
17
18    }
19
20    var auto = new Automovil();
21
22    console.log(auto.getColor());
23
24
25    })();
```

En este caso, *color* es atributo. Otro ejemplo es:

```
1  class Persona {
2      nombre:string;
3      decirMiNombre() {
4          console.log(this.nombre);
5      }
6  }
```

1.8.2 Constructores

Para generar una clase con constructor se tiene que utilizar la palabra reservada *constructor* como se muestra en el siguiente ejemplo:

```
1  (()=>{
2
3      class Automovil {
4
5        color:string;
6
7        constructor(color=""){
8
9          if(color==""){
10             this.color="rojo";
11         }else
12             this.color=color;
```

```
13
14        }
15
16        getColor(){
17
18            return this.color;
19
20        }
21        setColor(color:string){
22
23            this.color=color;
24
25        }
26
27    }
28
29    var auto = new Automovil();
30
31    console.log(auto.getColor());
32
33
34    })();
```

Otro ejemplo es el siguiente:

```
1  class Persona {
2      nombre:string;
3      apellido:string;
4      constructor(nuevoNombre:string, nuevoApellido:string) {
5          this.nombre=nuevoNombre;
6          this.apellido=nuevoApellido;
7      }
8  }
```

1.8.3 Visibilidad

La visibilidad permite indicar si los atributos o métodos son visibles únicamente dentro de la clase, desde una clase que tenga relación de herencia o bien fuera de la clase. Los modificadores de visibilidad en TypeScript son: *public*, *private* y *protected*.

El modificador *public* es el que viene por defecto en Typescript; cuando no se indica ningún modificador, el lenguaje pone por defecto este modificador. Lo que hace, es hacer visibles para todo el mundo el atributo o método. El modificador *private*, por otro lado, es lo opuesto a *public*.

Este modificador restringe la visibilidad de los atributos de la clase sólo a esta clase. *Protected* hace visible atributos entre clases padre e hijas, pero los hace no-visibles al resto del mundo. Un ejemplo usando los modificadores *private* y *public* son:

```
(()=>{

    class Automovil {

      private color:string;

      constructor(color=""){

        if(color==""){
            this.color="rojo";
        }else
            this.color=color;

      }

      public getColor(){

         return this.color;

      }
      public setColor(color:string){

         this.color=color;

      }

    }

    var auto = new Automovil();

    console.log(auto.getColor());


    })();
```

1.8.4 Herencia

La herencia permite definir una clase en términos de otra clase. En el caso de que una clase hija tenga un constructor, es obligatorio llamar al método super() antes de cualquier otra cosa. Lo que hace super precisamente, es llamar al constructor del padre, de esta forma, cualquiera que herede una clase se asegura de llamar al constructor de la clase padre, y así, inicializar todo lo que tenga que inicializar del padre. Por ejemplo:

```
1 class Automovil{
2     constructor() {
3     }
4 }
5 class Sentra extends Automovil {
6     constructor() {
7        super()
8     }
9 }
```

1.8.5 Interfaces

Las interfaces son un mecanismo de la programación orientada a objetos que trata de suplir la carencia de herencia múltiple. La mayoría de los lenguajes que implementan la orientación a objetos no ofrecen la posibilidad de definir una clase que extienda varias clases a la vez y sin embargo a veces es deseable. Ahí es donde entran las interfaces.

Una clase puede extender otra clase, heredando sus propiedades y métodos y declarar que implementa cualquier número de interfaces. La diferencia de las clases que extiendes con respecto a las interfaces es que las interfaces no contienen implementación de sus métodos, por lo que la clase que implementa una interfaz debe escribir el código de todos los métodos que contiene. Por este motivo, se dice que las interfaces son como un contrato, en el que se especifica las cosas que debe contener una clase para que pueda implementar una interfaz o cumplir el contrato declarado por esa interfaz.

```
1
2 (()=>{
3
4     interface AutoBase{
5
6         getColor():string;
7         getVelocidad():number;
8
9      }
10
```

```
11      class Automovil implements AutoBase{
12          private color:string;
13          private velocidad:number;
14          constructor(color:string,velocidad:number) {
15
16              this.color=color;
17              this.velocidad=velocidad;
18
19          }
20         public getColor():string{
21
22            return this.color;
23
24         }
25
26        public getVelocidad():number{
27
28            return this.velocidad;
29
30         }
31
32      }
33      class Rio extends Automovil {
34          constructor() {
35             super("rojo",5);
36          }
37
38      }
39
40      var auto = new Rio();
41      console.log(auto);
42
43
44    })();
```

Otro ejemplo:

```
1
2  (()=>{
3
4
5
6      interface Libro{
```

```
7
8           autor:string;
9           titulo:string;
10          institucion?:string;
11
12      }
13
14
15      const comedia :Libro ={
16
17          autor:"Anonimo",
18          titulo:"Mi jemplo"
19      }
20
21      console.log(comedia);
22
23  })();
```

1.9 Promesas

Las promesas dan la posibilidad de realizar alguna acción cuando una tarea asíncrona es finalizada, viene con el ecmascript 6. Cuando se crea una promesa, esta devuelve dos valores, dependiendo de si fue exitosa o no, por convención estos valores se llaman *resolve* (cuando fue exitoso) y *reject* (cuando fallo), –esto no significa que estas obligado a usar esas palabras-, pero por convención se utilizan así.

```
1   (()=>{
2
3       console.log("Inicio");
4
5       const prom1 = (n:number)=>{return new Promise((resolve,reject)=>{
6
7           var i:number;
8           var fact:number;
9           fact=1;
10          if(n>=0){
11
12              for(i=1;i<=n;i++){
13                  fact=fact*i;
14
15              }
```

```
            resolve(fact);

        }else{
            reject(-1);

        }

    });

}

    prom1(5)
        .then(function(mensaje){
                console.log("La operación fue exitosa:");
                console.log(mensaje);
         })
        .catch(function(err){
                console.log("La operación fue un fracaso:");
                console.warn(err);
         });

    prom1(-5)
        .then(mensaje=>console.log(mensaje))
        .catch(err=>console.warn(err));
    console.log("Fin");

})();
```

Otro ejemplo:

```

(()=>{


    const retirarDinero = (monto:number):Promise<number>=>{

        let dineroActual=1000;

        return new Promise((resolve,reject)=>{

            if(monto>dineroActual){
                reject("No hay suficiente dinero");
            }else{
```

```
14                  dineroActual-=monto;
15                  resolve(dineroActual);
16              }
17
18          });
19
20
21      }
22
23      retirarDinero(500).then(montoActual=>console.log(`Me queda ${
            montoActual}`)) .catch(err=>console.warn(err));
24
25      retirarDinero(1500).then(montoActual=>console.log(`Me queda ${
            montoActual}`)) .catch(err=>console.warn(err));
26
27
28  })();
```